October 1990

Report No. STAN-CS-90-1337

PB96-151337

A Simplifier for Untyped Lambda Expressions

by

Louis Galbiati and Carolyn Talcott

Department of Computer Science

Stanford University

Stanford, California 94305

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

2. REPORT DATE	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE

A Simplifier for Untyped Lambda Expressions

5. FUNDING NUMBERS

6. AUTHOR(S)

Louis Galbiati
Carolyn Talcott

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Department
Stanford University
Stanford, CA 94305

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

A short version in Proceedings of Conditional and Typed Rewriting Systems, Second International Workshop, June 1990

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release: distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Many applicative programming languages are based on the call-by-value lambda calculus. For these languages tools such as compilers, partial evaluations, and other transformation systems often make use of rewriting systems that incorporate some form of beta reduction. For purposes of automatic rewriting it is important to develop extensions of beta-value reduction and to develop methods for guaranteeing termination. This paper describes an extension of beta-value reduction and a method based on abstract interpretation for controlling rewriting to guarantee termination. The main innovations are (1) the use of rearrangement rules in combination with beta-value reduction to increase the power of the rewriting system and (2) the definition of a non-standard interpretation of expressions, the generates relation, as a basis for designing terminating strategies for rewriting.

14. SUBJECT TERMS	15. NUMBER OF PAGES 26
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

A Simplifier for Untyped Lambda Expressions

Louis Galbiati
Quickturn Systems, Inc.
Mountain View, CA

Carolyn Talcott
Stanford University
Stanford, CA
CLT@SAIL.STANFORD.EDU

Abstract

Many applicative programming languages are based on the call-by-value lambda calculus. For these languages tools such as compilers, partial evaluators, and other transformation systems often make use of rewriting systems that incorporate some form of beta reduction. For purposes of automatic rewriting it is important to develop extensions of beta-value reduction and to develop methods for guaranteeing termination. This paper describes an extension of beta-value reduction and a method based on abstract interpretation for controlling rewriting to guarantee termination. The main innovations are (1) the use of rearrangement rules in combination with beta-value reduction to increase the power of the rewriting system and (2) the definition of a non-standard interpretation of expressions, the *generates* relation, as a basis for designing terminating strategies for rewriting.

1. Introduction

The original motivation for this work came from a project to compile programs by transformation to continuation-passing style [Steele 1976]. This program transformation in its simplest form tends to introduce extraneous lambda-applications. Instead of complicating the transformation to avoid introducing these lambda-applications it seemed preferable to use it in conjunction with a general purpose simplifier. The idea being that such a simplifier could be shared by many automatic program manipulation tools as well as being useful in interactive program manipulation systems. For example, such a simplifier can be used for optimizing programs built by combining many components, since inlining procedure calls (call unfolding) and many peep-hole optimizations are instances of beta-reduction. It could also serve as a tool for building semantics directed compilers and partial evaluators.

Our simplifier is composed of a reduction system and a method for limiting application of reductions to insure termination. The basic reduction system can be used in combination with other control strategies and the analysis underlying our method for limiting reduction should work for variants of the reduction system.

The target language for our simplifier is that of the lambda calculus [Barendregt 1981]. The reduction system consists of the beta-value (beta-v) reduction

rule together with two rearrangement rules designed to create additional sites for the beta rule. The beta-v rule is the restriction of the standard beta conversion rule to applications in which the operand is a value expression, e.g. a variable, constant, or lambda abstraction. Thus $(\lambda x.f\ x)z$ is a beta-v reduction site (reducing to $f\ z$), while $(\lambda x.f\ x)(g\ z)$ is a beta reduction site but not a beta-v reduction site. The beta-v rule corresponds to call-by-value semantics for a programming language and [Plotkin 1975] shows that this rule is adequate to evaluate closed expressions. However there are many programs that are equivalent under a wide class of observations that cannot be proved equivalent in the lambda-v calculus. One example is the evaluated position context theorem: $C[e]$ is equivalent to $\texttt{let}\{x := e\}C[x]$ where C is any expression with a unique hole occurring in a position that will be evaluated before any other serious computation takes place [Talcott 1989]. The rearrangement rules of our reduction system are corollaries of this theorem expressing the fact that a let-binding (application of a lambda abstraction) occurring in the function position of an application or in the argument position of a application in which the function position contains a value can be moved outside of the application. Thus $(\texttt{let}\{f := g\ z\}\lambda x.f\ x)y$ rearranges to $\texttt{let}\{f := g\ z\}(\lambda x.f\ x)y$ and $(\lambda x.f\ x)\texttt{let}\{g := h\ z\}\lambda y.g\ y$ rearranges to $\texttt{let}\{g := h\ z\}(\lambda x.f\ x)(\lambda y.g\ y)$. Note that in both cases the expression before rearrangement has no beta-v reduction site, while the expression after rearrangement does have a beta-v reduction site. The rearrangement rules have the effect of moving expressions that intervene between a function and its argument to the outside. They define a canonical form in which functions are more likely to appear directly applied to their arguments.

The rearrangement rules by themselves form a confluent, terminating system. They are not derivable in the beta-v calculus and hence our reduction system is more powerful than one based purely on beta-v reduction.

[Moggi 1989] introduces the notion of computational monad as a framework for axiomatizing features of programming languages. Computational monads accomodate a wide variety of language features including assignment, exceptions, and control abstractions. An extension of the lambda-v calculus called the lambda-c calculus is presented and shown to be valid in all computational monads. Our rearrangement rules are derivable in the lambda-c calculus and thus are valid for any language whose semantics can be modeled as a computational monad.

Writing a simplifier based on rules that include beta reduction is made difficult by the fact that unrestricted application of these rules can lead to infinite reduction sequences. Thus a strategy is needed for limiting beta reduction. One possible strategy is to fix a maximum number of reduction steps and perform reductions at random until this limit is reached. This strategy has the disadvantage that it treats all reduction steps the same way, rather than favoring those which simplify the expression over those which wander aimlessly. A second strategy is to beta reduce a lambda-application $(\lambda x.e)v$ only if the bound variable x occurs free at most once in the body e or if the operand v is atomic. Call this the *reduces-size*

strategy. It guarantees that each beta reduction step decreases the size of the overall expression. This strategy can be overly conservative, since some expressions can be simplified only by first performing steps which increase the size of the expression, e.g. unfold and simplify. Note that neither of these strategies are confluent. This is obvious in the case of limiting the number of steps. To see this for the reduces size strategy we observe that for any lambda abstraction v $(\lambda x.\lambda z.(\lambda y.y(y\,z))x)v$ reduces to $(\lambda z.(\lambda y.y(y\,z))v)$ and to $(\lambda x.\lambda z.x(x\,z))v$.

In this paper we describe a new strategy, *statically limited* rewriting, in which we compute a subset **B** of lambda-nodes in the initial expression such that any rewriting of that expression is guaranteed to terminate if beta reduction is restricted to descendants of nodes in **B**. (The descendant relation is the natural relation between nodes in an expression and nodes in the result of rewriting that expression.)

We use a form of abstract interpretation (cf. [Abramsky and Hankin 1987]) to compute a suitable set **B**. First we define a non-standard interpretation of expressions, the *generates* relation **xgen** and the notion of a set of lambda nodes being an **xgen**-cycle. We then show that limiting reduction to descendants of a subset of lambda-nodes containing no **xgen**-cycle guarantees termination. Given an initial expression $e_{\mathtt{init}}$, **xgen** is a relation on reduction paths and pairs of lambda-nodes of $e_{\mathtt{init}}$ defined as follows. Let a and b be lambda nodes in the initial expression and let q be a reduction sequence beginning with $e_{\mathtt{init}}$. We say that a generates b in the final step of q (and write $\mathtt{xgen}(q,a,b)$), if the final step of q is a beta-v reduction at a site whose operator is a descendant of a, and this reduction step entails (in the case $a \neq b$) an increase in the number of descendants of b, or (in the case $a = b$) no decrease. We say a generates b along p if $\mathtt{xgen}(q,a,b)$ for some prefix q of p. A set of lambda nodes $a_0,\ldots,a_n$ in the initial expression is an **xgen**-cycle if, roughly, there is a reduction sequence along which a_i generates a_{i+1} for $i < n$ and a_n generates a_0.

For example consider the expression

$$(\lambda^1 x.x\,x)\,(\lambda^2 x.x\,x)$$

where the superscripts are used to associate names with lambda-nodes. Here there is a single reduction path along which 1 generates 2 and 2 generates 2. Limiting beta-reduction to descendants of node 1 guarantees termination (after one step!). As another example consider the expression

$$(\lambda^1 p.p\,p\,z)(\lambda^2 x.\lambda^3 y.\lambda^4 s.s\,x\,y)$$

For this expression there are reduction paths along which 1 generates $2,3,4$ and there are no other generates instances. Since there are no cycles all reduction sequences must terminate. Note that the reduces-size strategy mentioned earlier does not permit any reduction.

In general xgen can be an infinite relation. Thus we want to find a finite, computable approximation that serves the same purpose. Using the methodology of abstract interpretation we say that a relation together with a corresponding notion of cycle is a safe approximation to xgen if it preserves the "no-cycles implies termination" property. As a first step we define a binary relation gen on lambda nodes that is a safe finite approximation of xgen using the usual notion of cycle induced by a binary relation. gen is the set of pairs a, b such that for some reduction sequence q beginning with $e_{\texttt{init}}$, a generates b in the final step of q.

We are still not done, as we have no general (uniformly terminating) algorithm for computing gen. Instead we define a safe computable approximation gen′ of gen. The computation of gen′ is based on computing upper bounds to the sets of nodes in the initial expression whose descendants can occupy certain kinds of positions (cf. control flow analysis [Shivers 1988] and closure analysis [Bondorf 1990]) and on computing an upper bound to the set of lambda nodes in the initial expression that are "doublers", i.e. have a descendant with more than one free occurrence of the bound variable in the body. Then gen′ is roughly the set of all pairs (a, b) of lam nodes such that a is a doubler and there is some c such that a descendant of a is applied to a descendant c, and a descendant of b can become a subexpression of a descendant of c.

In addition to safety we need to show that the approximations we have defined are non-trivial (note that the complete binary relation on lambda nodes is a safe but useless approximation). In both of the examples above xgen, gen, and our computable approximation gen′ give rise to the same classification of cycles, and in particular gen and gen′ are non-trivial.

To summarize, given an expression to simplify, we proceed as follows: (i) compute gen′; (ii) choose a set **B** with no gen′-cycles; (iii) perform **B**-limited reduction until termination. Limited rewriting is in fact locally confluent. Thus we are free to apply the rules in whatever order we like; the final outcome will be the same.

Although usually less conservative than the reduces-size strategy, the new strategy is sometimes still overly conservative. A less conservative alternative strategy, *dynamically-limited* rewriting, is the following. Instead of computing gen′, we merely apply rules, accumulating a relation consisting of the pairs (a, b) such that a has generated b in some step of the rewriting so far, and disallowing any step which would cause this relation to contain a cycle. The alternative strategy guarantees termination but fails to preserve the confluence property. Nevertheless it may be the more appropriate strategy for a practical simplifier.

Our static and dynamic strategies have an analogue in two approaches to partial evaluation. The static strategy corresponds to the use of binding time analyis and other static analyses performed to determine which applications should be left to run time and which are to be carried out at partial-evaluation time (cf. [Jones, Sestoft, and Søndergaard 1989], and [Bondorf 1990]). The dynamic strategy is

more in the spirit of [Weise and Ruf 1990] where a call stack is maintained during partial evaluation and used for potential loop detection.

The rest of this paper is organized as follows. In Section 2 syntax and notation are described. In Section 3 the rewrite system is presented. In Section 4 the relation gen is introduced, two forms of limited rewriting are defined and shown to terminate, and it is shown that one form of limited rewriting is confluent while the other is not. In Section 5 we show that any superset of the relation gen is a safe approximation. The approximation gen′ is defined and proved safe. In Section 6 we discuss possible improvements and related work.

2. Syntax

We use standard lambda calculus syntax [Barendregt 1981]. To define and analyze reduction rules it is convenient to represent expressions as labeled trees where each node of the tree corresponds to an occurrence of a subexpression. In this section we define the set of expressions and their representation as labeled trees.

We assume given a countably infinite set **Var** of variables. Then the set **Exp** of expressions is the least set containing the variables and closed under lambda abstraction and application. That is, **Exp** is the least set satisfying the following equation.

$$\mathbf{Exp} = \mathbf{Var} \cup \lambda\mathbf{Var}.\mathbf{Exp} \cup \mathbf{Exp}\,\mathbf{Exp}$$

We let $x, x_0, \ldots$ range over **Var** and $e, e_0, \ldots$ range over **Exp**. Expressions of the form x, $\lambda x.\, e$, and $e_1\, e_2$ are called atomic expressions, abstractions, and applications, respectively. In an abstraction $\lambda x.\, e$, we call x the bound variable and e the body. In an application $e_1\, e_2$, we call e_1 the operator and e_2 the operand. We let **Vxp** be the set $\mathbf{Var} \cup \lambda\mathbf{Var}.\,\mathbf{Exp}$ of atomic expressions and abstractions; expressions in **Vxp** are called value expressions. We let $v, v_0, \ldots$ range over **Vxp**.

Free and bound variables are defined as usual and expressions identical up to alpha conversion we regard as indistinguishable. We write $e_1\{x := e_2\}$ for the result of substituting e_2 for all free occurrences of x in e_1. Here we assume that alpha variants are chosen "hygienically" so that no trapping of free variables occurs. $\texttt{let}\{x := e_0\}e_1$ abbreviates $(\lambda x.e_1)e_0$. We adopt the usual conventions for disambiguating written expressions, namely that (1) application associates left, so that $e_1\, e_2\, e_3$ is $(e_1\, e_2)\, e_3$, and (2) the body of an abstraction or `let` extends as far right as possible, so that $\lambda x.\, e_1\, e_2$ is $\lambda x.\, (e_1\, e_2)$. Parentheses may be used to override the default grouping as in $e_0(e_1\, e_2)$ or $(\lambda x.e_0)\, e_1$.

The tree structure of an expression is the abstract syntax tree modified to replace each bound variables by a pointer to the node in the tree corresponding to its binding lambda (cf. [deBruijn 1972]). Each node in the tree structure of an expression corresponds to a (unique) subexpression occurrence. Nodes are labeled by

the constructor of the corresponding subexpression and edges are labeled by component selectors. A pointer is represented by a path (sequence of edges) relative to a top-level expression. To make this precise we define selectors, locations, and tags as follows. A selector is an element of the set $\{\mathtt{L}, \mathtt{R}, \mathtt{B}\}$. Selectors name immediate subexpressions of an expression and label the edges of a tree. B names the body of an abstraction and L and R name the operator (left) and operand (right) components of an application. The set **Loc** of locations is the set of finite sequences with elements taken from the set of selectors.

$$\mathbf{Loc} = \{\mathtt{L}, \mathtt{R}, \mathtt{B}\}^*$$

Locations represent paths or nodes of a tree and are used to name occurrences of subexpressions. The set **Tag** of tags is defined by

$$\mathbf{Tag} = \{\mathtt{app}, \mathtt{lam}\} \cup \mathtt{atx}(\mathbf{Loc})$$

Tags label nodes of a tree. A nodes tag identifies the constructor of the corresponding subexpression and in the case of a bound variable the location of its binding abstraction.

We let $c, c_0, \ldots$ range over $\{\mathtt{L}, \mathtt{R}, \mathtt{B}\}$, $l, l_0, \ldots$ range over **Loc**, and $t, t_0, \ldots$ range over **Tag**. $\Box$ is the empty sequence and selectors are considered to be singleton sequences. We write $l.l'$ for the concatenation of the sequences l and l' and $l.c$ for the extension of l by c. If $l = l_0.l_1$ then l_0 is called a prefix of l.

For simplicity we will assume outermost expressions are closed (by adding lambdas if necessary). This is not a serious restriction, it just eliminates the need for a special case for free variables. For an outermost expression e, the locations, $\mathtt{locs}(e)$, the subexpression $(e)_l$ at location l and its tag $\mathtt{tag}(e, l)$ are defined by induction on the construction of e as follows.

(top) $\Box \in \mathtt{locs}(e)$ and $(e)_\Box = e$.

(app) If $l \in \mathtt{locs}(e)$ and $(e)_l = e_0\, e_1$ then $\mathtt{tag}(e, l) = \mathtt{app}$, $l.\mathtt{L}, l.\mathtt{R} \in \mathtt{locs}(e)$, $(e)_{l.\mathtt{L}} = e_0$, and $(e)_{l.\mathtt{R}} = e_1$.

(lam) If $l \in \mathtt{locs}(e)$ and $(e)_l = \lambda x.e_0$ then $\mathtt{tag}(e, l) = \mathtt{lam}$, $l.\mathtt{B} \in \mathtt{locs}(e)$, and $(e)_{l.\mathtt{B}} = e_0$.

(atx) If $l \in \mathtt{locs}(e)$, $(e)_l = x$, l' is a prefix of l, $(e)_{l'} = \lambda x.e'$, and l' is the longest such prefix of l then $\mathtt{tag}(e, l) = \mathtt{atx}(l')$

Let l be a location in e. If l has tag lam (i.e. $\mathtt{tag}(e, l) = \mathtt{lam}$), we say l is a lam-node of e. If l has tag app we say l is a app-node of e. If l has tag $\mathtt{atx}(l')$ we say that l is an atx-node bound at l' in e.

As an example let $e = \lambda f.\lambda x.f\, x$. The tree written as a term would be

$$\mathtt{lam}(\mathtt{B} : \mathtt{lam}(\mathtt{B} : \mathtt{app}(\mathtt{L} : \mathtt{atx}(\Box), \mathtt{R} : \mathtt{atx}(\mathtt{B}))))$$

where component selectors are made explicit using key-word argument syntax. Further we have

$$\texttt{locs}(e) = \{\square, \texttt{B}, \texttt{B.B}, \texttt{B.B.L}, \texttt{B.B.R}\}$$

$$\texttt{tag}(e,\square) = \texttt{lam} \qquad \texttt{tag}(e,\texttt{B}) = \texttt{lam} \qquad \texttt{tag}(e,\texttt{B.B}) = \texttt{app}$$

$$\texttt{tag}(e,\texttt{B.B.L}) = \texttt{atx}(\square) \qquad \texttt{tag}(e,\texttt{B.B.R}) = \texttt{atx}(\texttt{B})$$

The following basic facts about the tree structure of an expression are simple consequences of the definitions and will be used implicitly.

Lemma (tree.struc):

(app) If $l.\texttt{L} \in \texttt{locs}(e)$ or $l.\texttt{R} \in \texttt{locs}(e)$, then $l.\texttt{L} \in \texttt{locs}(e)$ and $l.\texttt{R} \in \texttt{locs}(e)$ and $\texttt{tag}(e,l) = \texttt{app}$.

(lam) If $l.\texttt{B} \in \texttt{locs}(e)$, then $\texttt{tag}(e,l) = \texttt{lam}$.

(atx) If $l \in \texttt{locs}(e)$ and $\texttt{tag}(e,l) = \texttt{atx}(l')$, then $\texttt{tag}(e,l') = \texttt{lam}$ and $l = l'.l_0$ for some l_0.

3. Reduction

An expression is simplified stepwise by applying one of three reduction rules.

(1) $(\lambda x.\, e_0)\, e_1\, e_2 \mapsto_1 (\lambda x.\, e_0\, e_2)\, e_1$ provided x is not free in e_2.

(2) $v\,((\lambda x.\, e_0)\, e_1) \mapsto_2 (\lambda x.\, v\, e_0)\, e_1$ provided x is not free in v.

(3) $(\lambda x.\, e_0)\, v \mapsto_3 e_0\{x := v\}$

The stepwise reduction relation $e \longrightarrow e'$ is the congruence closure of the union of the three reduction rules viewed as binary relations. That is, $e \longrightarrow e'$ just if for some $(r,l) \in \{1,2,3\} \times \mathbf{Loc}$, and some e_0, e_1 we have that $(e)_l = e_0$, $e_0 \mapsto_r e_1$, and e' is obtained from e by replacing the occurrence of e_0 at l by e_1. (Note that this is replacement, not substitution, and free variables of e_1 may be trapped by abstractions above l.) Pairs (r,l) for $r \in \{1,2,3\}$ and $l \in \mathbf{Loc}$ are called rule applications. We write $e \xrightarrow{(r,l)} e'$ to make the rule application explicit and we call l a site (in e) for application of rule r.

A reduction sequence is a sequence of stepwise reductions. We let p, p_0, ..., q, q_0, ... range over sequences of rule applications (r,l) and write $e \xrightarrow{p} e'$ if $p = (r_1,l_1),\ldots,(r_n,l_n)$ $e = e_0$, $e' = e_n$, and $e_{i-1} \xrightarrow{(r_i,l_i)} e_i$ for $1 \le i \le n$.

Rule 3 is the beta-v reduction rule [Plotkin 1975]. Rules 1 and 2, called left-rearrangement and right-rearrangement respectively, would be superfluous in a system with unlimited beta-reduction and beta-expansion. However with only call-by-value beta-reduction, these rules can create sites for application of rule 3 which

would not otherwise be created. Rearrangement merely rearranges the nodes in a tree, while beta-reduction may duplicate some subtrees and destroy others. The reduction rules preserve operational equivalence (cf. [Plotkin 75]). with respect to a call-by-value evaluator They are also valid in a wide range of extensions of the basic language including control abstractions [Talcott 1989] and memory operations [Mason and Talcott 1989a,b] and are valid for the λ_c theory of [Moggi 1989].

Theorem (Rearrangement is canonical): The reduction system generated by the rearrangement rules (the reflexive transitive congruence closure of $\mapsto_1 \cup \mapsto_2$) is terminating and confluent. Thus every expression has a unique normal form with respect to rearrangement.

Proof : What we must show is

(termination) Every sequence of rearrangements terminates.

(confluence) If e_0, e_1 are two distinct expressions that can be reached from an expression e by sequences of rearrangements then there is an expression e_2 that can be reached from both e_0 and e_1 by further sequences of rearrangements.

To prove termination, define the depth of a node as the number of `lam`'s it is below. In each rearrangement $e \longrightarrow e'$, the depth of the app node at the rearrangement site in e, and the depths of each node in one of its subtrees, increases by 1, while the depths of all other nodes remain constant. So the sum of the depths of all nodes increases in each step. But this sum is bounded by $n \times m$, where n is the number of nodes, and m the number of `lam` nodes, in e_1. So the sequence of rearrangements must be finite.

Since we have termination, to prove confluence it suffices to prove local confluence [Huet 1977]:

(local confluence) If e_0, e_1 are two distinct expressions that can be reached from an expression e by a one-step rearrangement then there is an expression e_2 that can be reached from both e_0 and e_1 by further sequences of rearrangements.

Instead of proving local confluence at this point we merely note that local confluence for rearrangement is a special case of local confluence of limited rewriting proved in the next section. □

In order to analyze properties of reduction sequences, we need to be able to trace the ancestry of nodes in an expression resulting from applying a sequence of reductions. For the direct application of a reduction rule $e \mapsto_r e'$ there is a natural predecessor in e of each node in e'. Consider an application of rule 1. Making the relevant tree structure explicit we have

$$e = \mathtt{app}^1(\mathtt{app}^2(\mathtt{lam}^3(x, e_0), e_1), e_2) \mapsto_1 \mathtt{app}^2(\mathtt{lam}^3(x, \mathtt{app}^1(e_0, e_2), e_1) = e'.$$

The predecessor of a node in the subexpression e_0, e_1, or e_2 of e' is the corresponding node the subexpression e_0, e_1, or e_2 of e. The predecessors of the remaining nodes

of e' are given by the superscripts. The predecessor function for applications of rule 2 or rule 3 is analogous. The precise definition is given below. For beta reduction this definition coincides with that of [Wadsworth 1978].

Definition (predecessor): For $e \xrightarrow{(r,l)} e'$ and $l' \in \mathtt{locs}(e')$ we define $\mathtt{pred}(e,(r,l),l')$, the (r,l)-predecessor of l' in e, as follows. If l is not a prefix of l' then $\mathtt{pred}(e,(r,l),l') = l'$. Otherwise pred is given by the following tables.

(1) If $r = 1$ and $(e)_l = (\lambda x.e_0)\,e_1\,e_2$ then $\mathtt{pred}(e,(1,l),l') = \hat{l}$ is given by

l'	$\hat{l}$	conditions
l	$l.\mathrm{L}$	
$l.\mathrm{L}$	$l.\mathrm{L.L}$	
$l.\mathrm{L.B}$	l	
$l.\mathrm{R}.l_1$	$l.\mathrm{L.R}.l_1$	$l_1 \in \mathtt{locs}(e_1)$
$l.\mathrm{L.B.L}.l_0$	$l.\mathrm{L.L.B}.l_0$	$l_0 \in \mathtt{locs}(e_0)$
$l.\mathrm{L.B.R}.l_2$	$l.\mathrm{R}.l_2$	$l_2 \in \mathtt{locs}(e_2)$

(2) if $r = 2$, $(e)_l = v\,((\lambda x.e_0)\,e_1)$ then $\mathtt{pred}(e,(2,l),l') = \hat{l}$ is given by

l'	$\hat{l}$	conditions
l	$l.\mathrm{R}$	
$l.\mathrm{L}$	$l.\mathrm{R.L}$	
$l.\mathrm{L.B}$	l	
$l.\mathrm{L.B.L}.l_v$	$l.\mathrm{L}.l_v$	$l_v \in \mathtt{locs}(v)$
$l.\mathrm{L.B.R}.l_0$	$l.\mathrm{R.L.B}.l_0$	$l_0 \in \mathtt{locs}(e_0)$
$l.\mathrm{R}.l_1$	$l.\mathrm{R.R}.l_1$	$l_1 \in \mathtt{locs}(e_1)$

(3) if $r = 3$, $(e)_l = (\lambda x.e_0)\,v$ then $\mathtt{pred}(e,(3,l),l') = \hat{l}$ is given by

l'	$\hat{l}$	conditions
$l.l_0$	$l.\mathrm{L.B}.l_0$	$l_0 \in \mathtt{locs}(e_0)$; $\mathtt{tag}(e, l.\mathrm{L.B}.l_0) \neq \mathtt{atx}(l.\mathrm{L})$
$l.l_0.l_v$	$l.\mathrm{R}.l_v$	$l_v \in \mathtt{locs}(v)$; $\mathtt{tag}(e, l.\mathrm{L.B}.l_0) = \mathtt{atx}(l.\mathrm{L})$

The following lemma is a direct consequence of the definitions. It expresses the key structural properties of reductions and points out the crucial distinction between rearrangements and beta reduction.

Lemma (pred): The predecessor function is 1-1 and onto except in the case of a rule 3 reduction where the application and abstraction nodes of the reduction site have no successors and nodes of the value may have zero or more successors.

The ancestor function anc generalizes the predecessor function to sequences of reduction steps mapping locations in the final expression of a reduction sequence to locations in the initial expression from which they derive.

Definition (ancestor): If $e \xrightarrow{p} e_p$ and $l \in \mathtt{locs}(e_p)$ then $\mathtt{anc}_e(p,l)$, the p-ancestor in e of l, is defined by induction on the length of p as follows.

(mt) $\mathtt{anc}_e(\Box, l) = l$

(nmt) If $p = p', (r, l')$ and $e \xrightarrow{p} e'$ then $\mathtt{anc}_e(p,l) = \mathtt{anc}_e(p', \mathtt{pred}(e', (r, l')))$.

If $\mathtt{anc}_e(p,l) = a$ then we say that l is a p-descendant of a.

The following lemma shows that, via the ancestor relation, tag types and binding relations are preserved by reduction.

Lemma (tag preservation): Let $e \xrightarrow{p} e'$, $l' \in \mathtt{locs}(e')$, and $\mathtt{anc}_e(p,l') = l$. If $\mathtt{tag}(e,l) \in \{\mathtt{app}, \mathtt{lam}\}$ then $\mathtt{tag}_e(p,l') = \mathtt{tag}(e,l)$. If $\mathtt{tag}(e,l) = \mathtt{atx}(l_0)$ then $\mathtt{tag}_e(p,l') = \mathtt{atx}(l'_0)$ where l'_0 is the (unique) location in e' such that l'_0 is a prefix of l' and $\mathtt{anc}_e(p,l'_0) = l_0$.

Proof : An easy induction on the length of the reduction sequence. The prefix requirement in the case of bound-variable tags distinguishes between copies of the value substituted into the body of a lambda expression in rule 3. □

4. Limited rewriting

In this section the relation gen is introduced, two forms of limited rewriting are defined and shown to terminate, and it is shown that one form of limited rewriting is confluent while the other is not. Finally we discuss limited rewriting as a basis for a practical rewrite-control strategy.

To simplify the definitions, for the remainder of the paper we fix an initial expression $e_{\mathtt{init}}$. $\mathbf{A}$ will denote the set of locations in $e_{\mathtt{init}}$ ($\mathbf{A} = \overline{\mathtt{locs}}(e_{\mathtt{init}})$) and $a, b, a_0, \ldots$ will range over $\mathbf{A}$. $\mathbf{A}_{\mathtt{lam}}$ will denote the set of lam locations in $e_{\mathtt{init}}$ ($\mathbf{A}_{\mathtt{lam}} = \{l \in \mathbf{A} \mid \mathtt{tag}(e_{\mathtt{init}}) = \mathtt{lam}\}$). Having fixed $e_{\mathtt{init}}$ we specialize the ancestor functions to $e_{\mathtt{init}}$ and omit the subscript. We let **Rseq** be the set of rule application sequences starting from $e_{\mathtt{init}}$, that is, sequences p such that $e_{\mathtt{init}} \xrightarrow{p} e$ for some e. For brevity, in situations where an expression is required a sequence p in **Rseq** may be used to denote the (unique) e such that $e_{\mathtt{init}} \xrightarrow{p} e$. In particular we will write $\mathtt{tag}(p,l)$ for $\mathtt{tag}(e,l)$.

4.1. The gen relation and limited rewriting

We begin by defining the *generates* relations xgen on $\mathbf{Rseq} \times \mathbf{A}_{\mathtt{lam}} \times \mathbf{A}_{\mathtt{lam}}$ and gen on $\mathbf{A}_{\mathtt{lam}} \times \mathbf{A}_{\mathtt{lam}}$.

Let a and b be lam nodes in $\mathbf{A}_{\mathtt{lam}}$ and let q be a rule-application sequence in **Rseq**. We say that a generates b in the final step of q (and write $\mathtt{xgen}(q,a,b)$), if the final step of q is a rule-3 reduction at a site whose operator is a descendant of a, and this final step entails (in the case $a \neq b$) an increase in the number of descendants of b, or (in the case $a = b$) no decrease.

Definition (xgen): $\mathtt{xgen}(q,a,b)$ just if $a,b \in \mathbf{A}_{\mathtt{lam}}$, $q \in \mathbf{Rseq}$, and there are p,e,e',l such that $q = p.(3,l)$ and (i-iii) hold.

(i) $e_{\mathtt{init}} \xrightarrow{p} e \xrightarrow{(3,l)} e'$

(ii) $\mathtt{anc}(p, l.\mathrm{L}) = a$

(iii) $n_b < n'_b$ if $a \neq b$ and $n_b \leq n'_b$ if $a = b$; where n_b is the number of locations l' in e such that $\mathtt{anc}(p,l') = b$ and n'_b is the number of locations l' in e' such that $\mathtt{anc}(p.(3,l),l') = b$.

We say that a generates b (and write $\mathtt{gen}(a,b)$) if a generates b in some step of some reduction sequence beginning with $e_{\mathtt{init}}$.

Definition (gen): $\mathtt{gen}(a,b)$ just if there is some $q \in \mathbf{Rseq}$ for which $\mathtt{xgen}(q,a,b)$.

We now define two forms of limited rewriting.

Definition (R-limited rewriting): Given a relation $\mathbf{R}$ on $\mathbf{A}_{\mathtt{lam}} \times \mathbf{A}_{\mathtt{lam}}$, we define an $\mathbf{R}$-limited rewriting to be any reduction sequence $e_{\mathtt{init}} \longrightarrow e_1 \longrightarrow \ldots$ starting with $e_{\mathtt{init}}$, and satisfying the restriction that a step in which some node a generates some node b is allowed only if $(a,b) \in \mathbf{R}$. That is, if $e_{\mathtt{init}} \xrightarrow{p} e \xrightarrow{(3,l)} e'$ is an initial segment of such a sequence and $\mathtt{xgen}(p.(3,l),a,b)$, then $(a,b) \in \mathbf{R}$.

Definition (B-limited rewriting): Given a subset $\mathbf{B}$ of $\mathbf{A}_{\mathtt{lam}}$ we define a $\mathbf{B}$-limited rewriting to be any reduction sequence $e_{\mathtt{init}} \longrightarrow e_1 \longrightarrow \ldots$ starting with $e_{\mathtt{init}}$, and satisfying the restriction that a beta reduction step is allowed only if the operator is a descendant of a location in $\mathbf{B}$. Thus if $e_{\mathtt{init}} \xrightarrow{p} e \xrightarrow{(3,l)} e'$ is an initial segment of such a sequence then $\mathtt{anc}(p, l.\mathrm{L}) \in \mathbf{B}$.

4.2. Termination of limited rewriting

In this subsection we show that under suitable conditions each of the two forms of limited rewriting is guaranteed to terminate. We say that a binary relation R on a set X has no cycles if there is no sequence $x_0, \ldots, x_n$ of elements of X such that $x_0 = x_n$ and $R(x_i, x_{i+1})$ for $0 \leq i < n$.

Theorem (R-limited rewriting terminates): Let $\mathbf{R}$ be a relation on $\mathbf{A}_{\mathtt{lam}} \times \mathbf{A}_{\mathtt{lam}}$ with no cycles. Then any $\mathbf{R}$-limited rewriting must be finite.

Proof : Let $e_{\mathtt{init}} \longrightarrow e_1 \longrightarrow \ldots$ be an **R**-limited rewriting, and let p be the corresponding (possibly infinite) sequence of (r,l) pairs. Let $\mathbf{R}_p$ be the set of (a, b) pairs such that a generates b in some step of this rewriting, that is,

$$\mathbf{R}_p = \{(a,b) \in \mathbf{A}_{\mathtt{lam}} \times \mathbf{A}_{\mathtt{lam}} \mid (\exists q, q_1 \in \mathbf{Rseq})(p = q.q_1 \wedge \ \mathtt{xgen}(q,a,b))\}.$$

Since **R** has no cycles, neither does $\mathbf{R}_p$, and we can linearly order the elements of $\mathbf{A}_{\mathtt{lam}}$ as a row $a_1, \ldots, a_n$ such that during this rewriting each element of the row generates only elements to the right of that element. That is, for $a_j, a_k \in \mathbf{A}_{\mathtt{lam}}$ and q a finite prefix of p, if $\mathtt{xgen}(q, a_j, a_k)$ then $j < k$.

Define the rearrangement potential for an expression e to be the number of steps in the longest sequence of rearrangements beginning with e. Since rearrangement is terminating the rearrangement potential is always a natural number, and decreases with any rearrangement step.

For each expression e_i in $e_{\mathtt{init}} \longrightarrow e_1 \longrightarrow \ldots$, let τ_i be the $(n+1)$-tuple of natural numbers whose first n components are the numbers of descendants in e_i of $a_1, \ldots, a_n$, respectively, and whose last component is the rearrangement potential of e_i. We show that the sequence of tuples $\tau_{\mathtt{init}}, \tau_1, \ldots$ is in lexicographically decreasing order. Hence both the sequence $\tau_{\mathtt{init}}, \tau_1, \ldots$ and the sequence $e_{\mathtt{init}} \longrightarrow e_1 \longrightarrow \ldots$ must be finite.

Suppose $e_i \longrightarrow e_{i+1}$ is a rearrangement step. Since for rearrangements the predecessor function is one-to-one and onto, τ_i and τ_{i+1} are equal in their first n components. Since the rearrangement potential decreases in a rearrangement step, the last component of τ_{i+1} is less than that of τ_i. Suppose $e_i \longrightarrow e_{i+1}$ is a beta-value reduction step. Then the operator at the reduction site must be a descendant of a node a_j in $\mathbf{A}_{\mathtt{lam}}$. The jth component of τ_{i+1} must be less than that of τ_i, and no preceding component of τ_{i+1} can be greater than the corresponding component of τ_i. Otherwise, for q the prefix of p corresponding to $e_{\mathtt{init}} \longrightarrow \ldots \longrightarrow e_{i+1}$ and k the offending position at or before position j, we would have $\mathtt{xgen}(q, a_j, a_k)$, violating the condition by which the elements $a_1, \ldots, a_n$ were ordered. □

BeginNote

From the proof we see that (**R-limited rewriting terminates**) holds for any extension of the beta-v rule by the addition of a terminating collection of rules with the property that application of one of these rules never increases the number of descendants of a node.

EndNote

Corollary (B-limited rewriting terminates): Let **B** be a subset of $\mathbf{A}_{\mathtt{lam}}$ such that the restriction $\mathtt{gen}_{\mathbf{B}}$ of gen to **B** has no cycles. Then any **B**-limited rewriting must be finite.

Proof : Any **B**-limited rewriting is $\mathtt{gen}_{\mathbf{B}}$-limited (since if there is a step which makes a rewriting not $\mathtt{gen}_{\mathbf{B}}$-limited, then the step must be a beta reduction step

and the operator at the reduction site must be a descendant of an element of **B**, so the rewriting is not **B**-limited.) Since $\mathtt{gen}_{\mathbf{B}}$ has no cycles, any **B**-limited rewriting must be finite by the preceding theorem. □

4.3. Confluence of limited rewriting

In the previous section we showed that for certain subsets **B** of $\mathbf{A}_{\mathtt{lam}}$, **B**-limited rewriting terminates. In this section we show that for *any* subset **B** of $\mathbf{A}_{\mathtt{lam}}$, **B**-limited rewriting is locally confluent. **R**-limited rewriting, however, is not confluent.

Theorem (B-limited rewriting is locally confluent): If **B** is any subset of $\mathbf{A}_{\mathtt{lam}}$ then **B**-limited rewriting is locally confluent. That is, if $e_{\mathtt{init}} \xrightarrow{p} e \xrightarrow{(r_k,l_k)} e_k$ is a **B**-limited rewriting for $k \in \{\alpha,\beta\}$ then we can find p_k and e' such that $e_{\mathtt{init}} \xrightarrow{p} e \xrightarrow{(r_k,l_k)} e_k \xrightarrow{p_k} e'$ is a **B**-limited rewriting for $k \in \{\alpha,\beta\}$.

Proof: Assume $e_{\mathtt{init}} \xrightarrow{p} e \xrightarrow{(r_k,l_k)} e_k$ is a **B**-limited rewriting for $k \in \{\alpha,\beta\}$. We want to find p_k and e' such that $e_{\mathtt{init}} \xrightarrow{p} e \xrightarrow{(r_k,l_k)} e_k \xrightarrow{p_k} e$ is a **B**-limited rewriting for $k \in \{\alpha,\beta\}$. Note that if $l \in \mathtt{locs}(e)$ is a site for **B** application of rule 3 and l' is a descendant of l in e_k then l' is a site for **B** application of rule 3 in e_k. If l_α is not a prefix of l_β and l_β is not a prefix of l_α then l_α is a site for rule r_α in e_β, l_β is a site for rule r_β in e_α, and applying the rules in either order gives the same result (call it e'). Thus $e_{\mathtt{init}} \xrightarrow{p} e \xrightarrow{(r_k,l_k)} e_k \xrightarrow{(r_{\bar{k}},l_{\bar{k}})} e'$ is a **B**-limited rewriting for $k \in \{\alpha,\beta\}$ and $\bar{k}$ the opposite of k. Thus without loss of generality we may assume that l_α is a prefix of l_β and consider three cases according to whether r_α is 1, 2, or 3.

Case 1: Let $(e)_{l_\alpha} = (\lambda x.e_0)\,e_1\,e_2$. If l_β is a location in e_0, e_1, or e_2 then application of the two rules commutes.

$$e \xrightarrow{(r_\alpha,l_\alpha)} e_\alpha \xrightarrow{(r_\beta,l'_\beta)} e' \quad \text{and} \quad e \xrightarrow{(r_\beta,l_\beta)} e_\beta \xrightarrow{(r_\alpha,l_\alpha)} e'$$

where $\mathtt{pred}(e_\alpha, l'_\beta) = l_\beta$ and we are done. Otherwise (by the form of the rules) we have $l_\beta = l_\alpha.\mathrm{L}$ and $r_\beta = 2$ or $r_\beta = 3$.

Case 1.2: $e_1 = (\lambda y.e_3)\,e_4$

$$\begin{aligned}
&(\lambda x.e_0)\,e_1\,e_2 \mapsto_1 (\lambda x.e_0\,e_2)\,e_1 \\
&\quad = (\lambda x.e_0\,e_2)\,((\lambda y.e_3)\,e_4) \mapsto_2 (\lambda y.(\lambda x.e_0\,e_2)\,e_3)\,e_4 \\
&(\lambda x.e_0)\,((\lambda y.e_3)\,e_4)\,e_2 \xrightarrow{(2,\mathrm{L})} (\lambda y.(\lambda x.e_0)\,e_3)\,e_4\,e_2 \\
&\quad \mapsto_1 (\lambda y.(\lambda x.e_0)\,e_3\,e_2)\,e_4 \xrightarrow{(1,\mathrm{L.B})} (\lambda y.(\lambda x.e_0\,e_2)\,e_3)\,e_4
\end{aligned}$$

Case 1.3: $e_1 = v_1 \in \mathbf{Vxp}$

$$(\lambda x.e_0)\, v_1\, e_2 \mapsto_1 (\lambda x.e_0\, e_2)\, v_1 \mapsto_3 e_0\{x := v_1\}\, e_2$$

$$(\lambda x.e_0)\, v_1\, e_2 \xrightarrow{(3,\mathrm{L})} e_0\{x := v_1\}\, e_2$$

Case 2: Let $(e)_{l_\alpha} = v\,((\lambda x.e_0)\, e_1)$. Again if l_β is a location in v, e_0, or e_1 then application of the two rules commutes (modulo relocation) and we are done. Otherwise (by the form of the rules) we have $l_\beta = l_\alpha.\mathrm{R}$ and $r_\beta = 2$ or $r_\beta = 3$.

Case 2.2: $e_1 = (\lambda y.e_2)\, e_3$

$$v\,((\lambda x.e_0)\, e_1) \mapsto_2 (\lambda x.v\, e_0)\, e_1$$

$$= (\lambda x.v\, e_0)\,((\lambda y.e_2)\, e_3) \mapsto_2 (\lambda y.(\lambda x.v\, e_0)\, e_2)\, e_3$$

$$v\,((\lambda x.e_0)\,((\lambda y.e_2)\, e_3)) \xrightarrow{(2,\mathrm{L})} v\,((\lambda y.(\lambda x.e_0)\, e_2)\, e_3)$$

$$\mapsto_2 (\lambda y.v\,((\lambda x.e_0)\, e_2))\, e_3 \xrightarrow{(2,\mathrm{L.B})} (\lambda y.(\lambda x.v\, e_0)\, e_2)\, e_3$$

Case 2.3: $e_1 = v_1 \in \mathbf{Vxp}$

$$v\,((\lambda x.e_0)\, v_1) \mapsto_2 (\lambda x.v\, e_0)\, v_1 \mapsto_3 v\, e_0\{x := v_1\} \qquad \text{\% since } x \notin \mathrm{FV}(v)$$

$$v\,((\lambda x.e_0)\, v_1) \mapsto_3 v\,(e_0\{x := v_1\})$$

Case 3: We use the following standard lemmas.

(i) $e \xrightarrow{(r,l)} e' \Rightarrow e\{x := v\} \xrightarrow{(r,l)} e'\{x := v\}$

(ii) $v \xrightarrow{(r,l)} v' \Rightarrow e\{x := v\} \xrightarrow{(r,l.l_1),\ldots,(r,l.l_n)} e\{x := v'\}$ where $l_1, \ldots, l_n$ is a list of the locations of free occurrences of x in e.

Let $(e)_{l_\alpha} = \lambda x.e_0\, v$. Then l_β must be a location in e_0 or v and the result follows from the lemmas (i) and (ii) respectively. □

Corollary (B-limited rewriting is canonical): Each expression $e_{\mathtt{init}}$ has a unique simplified form with respect to B-limited rewriting for any $\mathbf{B} \subseteq \mathbf{A}_{\mathtt{lam}}$ such that $\mathbf{gen}_{\mathbf{B}}$ contains no cycles.

BeginNote

One might suppose that **R**-limited rewriting is canonical for any $\mathbf{R} \subseteq \mathbf{A}_{\mathtt{lam}} \times \mathbf{A}_{\mathtt{lam}}$ with no cycles. This conjecture is false. For example, take

$$e_{\mathtt{init}} = (\lambda^1 z.(\lambda^2 x.x\, x)(\lambda^3 p.z))(\lambda^4 w.w)$$

and $\mathbf{R} = \{(2,3)\}$. There are two choices for the first step of **R**-limited reduction, and the resulting expressions have no common reachable expression.

EndNote

4.4. Strategies for controlling rewriting

The results of this section suggest the following strategies for controlling rewriting.

(1: Statically-limited) Compute gen, choose a maximal subset **B** of $\mathbf{A}_{\mathtt{lam}}$ with no gen-cycles, and perform **B**-limited rewriting until termination.

(2: Dynamically-limited) Instead of computing gen, merely apply rules, accumulating information about the xgen relation as the set of pairs (a, b) such that a has generated b in some step of the rewriting so far, and disallowing any step which would cause this relation to contain a cycle. Since any reduction sequence generated by this method is **R**-limited for some **R** with no cycles, no infinite reduction sequence can be generated.

The first strategy has some obvious advantages. First, it is fully specified in the sense that it terminates with the same final result regardless of the order in which rules are applied. This means that it is simpler to analyze. Another advantage of strategy (1) is that it does not require computing generation pairs (a, b) at each beta reduction step. In practice, since we have no algorithm for computing gen, strategy (1) will be implemented using some safe approximation gen′ of gen. One such approximation is described in the next section.

Let us say that one rewrite-control strategy is *always as powerful* as another if every reduction sequence allowed by the first is allowed by the second. Otherwise we say that the first is *sometimes less powerful* than the second (and the second *sometimes more powerful* than the first). It is interesting to compare the power of strategies (1) and (2) with that of the reduces-size strategy mentioned in the introduction.

Both of the strategies (1) and (2) are sometimes more powerful than the reduces-size strategy (for example consider the second example given in the introduction). Reduces-size rewriting is identical with **R**-limited rewriting with **R** the empty relation. So strategy (2) is always as powerful as the reduces-size strategy.

Strategy (1) is sometimes less powerful than the reduces-size strategy. For example if the initial expression is

$$\mathtt{let}^1\{f := \lambda^3 y.\mathtt{let}^4\{x := y\}\mathtt{let}^5\{w := x\}w\,w\}f\,\lambda^2 z.f$$

then gen includes the cycle $4 \longrightarrow 4$. (To see this, reduce the application of 1; then reduce leftmost applications of 3, 5, and 4. Node 4 generates itself in the last step.) This means that our choice of **B** for statically-limited rewriting cannot include node 4. Thus, statically-limited rewriting will not allow reduction of 4-applications. However the reduces-size strategy allows reduction of a 4-application as the first step.

5. Estimating the gen relation

We defined a relation gen on $\mathbf{A}_{\mathtt{lam}}$ the set of lambda nodes of a given initial expression $e_{\mathtt{init}}$ and showed that if a subset $\mathbf{B}$ of $\mathbf{A}_{\mathtt{lam}}$ contains no gen-cycles then $\mathbf{B}$-limited rewriting from $e_{\mathtt{init}}$ terminates. As it stands, this result is of little use, since we have no algorithm to compute the relation gen for an arbitrary expression $e_{\mathtt{init}}$. Instead we will define a computable relation gen$'$ which is a safe approximation to gen. We say that gen$'$ is a safe approximation to gen if whenever a subset $\mathbf{B}$ of $\mathbf{A}_{\mathtt{lam}}$ has no gen$'$ cycles then it has no gen cycles. Thus we can safely use gen$'$ to choose the subset $\mathbf{B}$ for limited rewriting.

Lemma (gen.safe): If gen$'$ is a relation on $\mathbf{A}_{\mathtt{lam}} \times \mathbf{A}_{\mathtt{lam}}$ that is a superset of gen (gen$(a,b) \Rightarrow$ gen$'(a,b)$) then gen$'$ is a safe approximation of gen.

In this section we define a computable relation gen$'$ that is a superset of gen for any given $e_{\mathtt{init}}$. The development for our algorithm for calculating gen$'$ was based on the following intuitions.

(1) Nodes (atomic expression nodes, application nodes, and lambda nodes) are considered to maintain their identity as reduction proceeds.

(2) Each application node has two hooks, and each lambda node one hook, to which the root nodes of subexpressions are attached. During reduction the node attached to a given hook may be removed and a new node attached.

(3) One can simultaneously determine for every hook an upper bound on the set of `lam` or `atx` nodes which can ever become attached to that hook, in the following way. We know the node initially attached to each hook. There are only two ways a given hook can get a new node: (a) when a lambda-application $\mathtt{app}^1(\mathtt{lam}^2(e1), e2)$ is reduced, each hook within $\mathtt{lam}^2(e1)$ to which a variable-node bound by $\mathtt{lam}^2$ is attached gets (a copy of) the node currently attached to the right-hand hook of $\mathtt{app}_1$. (b) when the above-mentioned reduction occurs, the hook to which the node $\mathtt{app}^1$ is attached, gets the node attached to the hook of $\mathtt{lam}^2$. To simultaneously build the upper-bound set of nodes for every hook, we proceed as follows. Each node-set initially contains zero or one node. If there is an app node whose left hook node-set contains $\mathtt{lam}^i$ and whose right hook node-set contains node n then add n to the node-set of each hook whose node-set contains an `atx` node bound by $\mathtt{lam}^i$, and to the node-set of the hook to which this app is originally attached, add all elements in the node-set of the hook of $\mathtt{lam}^i$.

(4) By analogous methods we can determine upper bounds for the set of `lam` nodes which are "doublers" (a `lam` node with some descendent that contains more than one occurrence of the bound variable in the body) and for the set of pairs (n_1, n_2) of `atx` or `lam` nodes such that node n_2 can occupy a position at or below n_1 (so that attaching n_1 to a given hook "can bring" node n_2 along with it). Finally we compute gen$'$ as the set of all pairs (n_1, n_2) of `lam` nodes such

that n_1 is a doubler and n_1 is in the node-set of the left hook of an app node whose right hook node set includes a node which can bring n_2.

To compute gen′ we first define auxiliary relations get, doubler, and canbring expressing the key features in the clauses of the definition of gen and show that gen is approximated by a simple combination of these relations. We then define computable relations get′, doubler′, and canbring′ that are safe approximations (supersets) of get, doubler, and canbring respectively. gen′ is then defined to be the corresponding combination of the approximations to the auxiliary relations.

As motivation we begin with a lemma (**gen.char**) characterizing gen. This lemma states that $\mathtt{gen}(a,b)$ holds just if there is some rewriting e of $e_{\mathtt{init}}$ with a site for application of the beta-v rule such that the ancestor of the abstraction component is a, the bound variable of that abstraction occurs at least twice in the body, and there is a location within the value component with ancestor b.

Lemma (gen.char): $\mathtt{gen}(a,b)$ just if $a,b \in \mathbf{A}_{\mathtt{lam}}$ and there are p,l,e,e',l_0,l_1,l_2 such that

(i) $e_{\mathtt{init}} \xrightarrow{p} e \xrightarrow{(3,l)} e'$,

(ii) $\mathtt{anc}(p,l.\mathrm{L}) = a$ and $\mathtt{anc}(p,l.\mathrm{R}.l_0) = b$,

(iii) $l_1 \neq l_2$ and $\mathtt{tag}(p,l.\mathrm{L}.l_1) = \mathtt{tag}(p,l.\mathrm{L}.l_1) = \mathtt{atx}(l.\mathrm{L})$.

Proof : The if direction is trivial. For the onlyif direction, assume $\mathtt{gen}(a,b)$ and let p,l,e,e' be such that $e_{\mathtt{init}} \xrightarrow{p} e \xrightarrow{(3,l)} e'$, $\mathtt{anc}(p,l.\mathrm{L}) = a$, and (in the case $a = b$) $n_b \leq n'_b$ or (in the case $a \neq b$) $n_b < n'_b$; where n_b is the number of locations l' in e such that $\mathtt{anc}(p,l') = b$ and n'_b is the number of locations l' in e' such that $\mathtt{anc}(p,l') = b$. If there is no l_0 such that $\mathtt{anc}(p,l.\mathrm{R}.l_0) = b$ or if $\mathtt{tag}(p,l.\mathrm{L}.l_1) = \mathtt{tag}(p,l.\mathrm{L}.l_2) = \mathtt{atx}(l.\mathrm{L})$ implies $l_1 = l_2$ then (since the subexpression at b is not a variable) $b \neq a$ implies $n_b = n'_b$ and $b = a$ implies $n_b > n'_b$. Thus we can find l_0,l_1,l_2 such that $\mathtt{anc}(p,l.\mathrm{R}.l_0) = b$, $l_1 \neq l_2$, and $\mathtt{tag}(p,l.\mathrm{L}.l_1) = \mathtt{tag}(p,l.\mathrm{L}.l_2) = \mathtt{atx}(l.\mathrm{L})$. □

get is a relation on $\mathbf{A} \times \{\mathrm{L},\mathrm{R},\mathrm{B}\} \times \mathbf{A}$ such that for locations a, b in the initial expression, $\mathtt{get}(a,c,b)$ means that there is a rewriting p of $e_{\mathtt{init}}$ such that there is a p-descendant of a with a p-descendant of b immediately below it along a c edge.

Definition (get): $\mathtt{get}(a,c,b)$ just if $a,b \in \mathbf{A}$, $c \in \{\mathrm{L},\mathrm{R},\mathrm{B}\}$, and there is some p,l,e such that $e_{\mathtt{init}} \xrightarrow{p} e$, $\mathtt{anc}(p,l) = a$, and $\mathtt{anc}(p,l.c) = b$.

canbring is a relation on $\mathbf{A} \times \mathbf{A}$ such that if $\mathtt{canbring}(a_1,a_2)$ then a_1 and a_2 are lam-nodes and there is a rewriting p of $e_{\mathtt{init}}$ such that there is a p-descendant of a_1 which is in a "potential operand" location (a location ending with R or B), and which has p-descendant of a_2 below it.

Definition (canbring): For $a_1,a_2 \in \mathbf{A}$ $\mathtt{canbring}(a_1,a_2)$ just if $\mathtt{tag}(e_{\mathtt{init}},a_1) = \mathtt{tag}(e_{\mathtt{init}},a_2) = \mathtt{lam}$ and there are p, l, c, l_0, l_1 such that $c \in \{\mathrm{R},\mathrm{B}\}$ such that $\mathtt{anc}(p,l.c) = a_1$ and $\mathtt{anc}(p,l.c.l_0) = a_2$.

For a node a in the initial expression, $\texttt{doubler}(a)$ means that a is a lam-node and that there is a rewriting p from $e_{\texttt{init}}$ such that a p-descendant of a has more than one occurrence of its bound variable in its body.

Definition (doubler): For $a \in \mathbf{A}$, $\texttt{doubler}(a)$ just if there are p, l, l_1, l_2 such that $l_1 \neq l_2$, $\texttt{anc}(p, l) = a$, $\texttt{tag}(p, l) = \texttt{lam}$, $\texttt{tag}(p, l.l_1) = \texttt{tag}(p, l.l_2) = \texttt{atx}(l)$.

Approximations to gen can be factored into approximations of get, canbring, and doubler using the following theorem.

Theorem (gen.approx): If $\texttt{gen}(a, b)$ then $\texttt{doubler}(a)$ and there are $a_0, a_1 \in \mathbf{A}$ such that $\texttt{get}(a_0, \texttt{L}, a)$, $\texttt{get}(a_0, \texttt{R}, a_1)$, and $\texttt{canbring}(a_1, b)$.

Proof : A direct consequence of **(gen.char)**. □

5.1. Approximating the factors of gen

The approximations get′, canbring′, and doubler′ are defined inductively as the least relations satisfying certain conditions (sets of clauses). The clauses were determined systematically by seeing what was needed to carry through a proof of safeness by induction on the rewriting p that occurs in the definitions of the corresponding exact relations. The base case is $p = \square$ and the corresponding clause was obtained by instantiating the formula defining the exact relation with $p = \square$. For p non-empty we consider the last rule applied, assume safeness for shorter rewritings, and analyze the possible relations between the location of the rule application and the locations mentioned in the definition of the exact relation. The labels of the clauses in the definitions of get′, canbring′, and doubler′ below reflect this case analysis which is given in more detail in the proofs of safeness. For the definitions we need one additional auxiliary relation isval on $\mathbf{A}$ which is true for value locations in the initial expression.

Definition (isval): $\texttt{isval}(a) \Leftrightarrow (e_{\texttt{init}})_a \in \mathbf{Vxp}$

Lemma (isval): $\texttt{isval}(a)$ just if $\texttt{tag}(e_{\texttt{init}}, a) = \texttt{lam}$ or $\texttt{tag}(e_{\texttt{init}}, \bar{a})^{-} = \texttt{atx}(b)$ for some b in $\mathbf{A}_{\texttt{lam}}$.

5.1.1. Approximating get

Definition (getp): get′ is the least relation on $\mathbf{A} \times \{\texttt{L}, \texttt{R}, \texttt{B}\} \times \mathbf{A}$ such that

(mt) $\texttt{get}'(a, c, a.c)$

(1.1) $\texttt{get}'(a, c, a_0) \wedge \texttt{get}'(a_0, \texttt{L}, b) \wedge \texttt{get}'(b, \texttt{L}, a_1) \wedge \texttt{tag}(\square, a_1) = \texttt{lam}$
$\Rightarrow \texttt{get}'(a, c, b)$

(1.2) $\texttt{get}'(b, \texttt{L}, a_0) \wedge \texttt{get}'(a_0, \texttt{L}, a) \wedge \texttt{tag}(\square, a) = \texttt{lam} \Rightarrow \texttt{get}'(a, \texttt{B}, b)$

(1.3) $\texttt{get}'(a, \texttt{L}, a_0) \wedge \texttt{get}'(a_0, \texttt{L}, a_1) \wedge \texttt{get}'(a_1, \texttt{B}, b) \Rightarrow \texttt{get}'(a, \texttt{L}, b)$

(2.1) $\texttt{get}'(a, c, a_0) \wedge \texttt{get}'(a_0, \texttt{R}, b) \wedge \texttt{get}'(a_0, \texttt{L}, a_1) \wedge \texttt{isval}(a_1)$

$$\wedge\ \mathtt{get}'(b,\mathrm{L},a_2) \wedge \mathtt{tag}(\square,a_2) = \mathtt{lam} \Rightarrow \mathtt{get}'(a,c,b)$$

(2.2) $\mathtt{get}'(b,\mathrm{L},a_0) \wedge \mathtt{isval}(a_0) \wedge \mathtt{get}'(b,\mathrm{R},a_1) \wedge \mathtt{get}'(a_1,\mathrm{L},a) \wedge \mathtt{tag}(\square,a) = \mathtt{lam}$
$\Rightarrow \mathtt{get}'(a,\mathrm{B},b)$

(2.3) $\mathtt{get}'(a,\mathrm{R},a_0) \wedge \mathtt{get}'(a_0,\mathrm{L},a_1) \wedge \mathtt{get}'(a_1,\mathrm{B},b) \wedge \mathtt{get}'(a,\mathrm{L},a_2) \wedge \mathtt{isval}(a_2)$
$\Rightarrow \mathtt{get}'(a,\mathrm{R},b)$

(3.1) $\mathtt{get}'(a,c,a_0) \wedge \mathtt{get}'(a_0,\mathrm{L},a_1) \wedge \mathtt{get}'(a_1,\mathrm{B},b) \wedge \mathtt{get}'(a_0,\mathrm{R},a_2) \wedge \mathtt{isval}(a_2)$
$\Rightarrow \mathtt{get}'(a,c,b)$

(3.2) $\mathtt{get}'(a,c,a_0) \wedge \mathtt{get}'(a_0,\mathrm{R},b) \wedge \mathtt{isval}(b) \wedge \mathtt{get}'(a_0,\mathrm{L},a_1) \wedge \mathtt{get}'(a_1,\mathrm{B},a_2)$
$\wedge\ \mathtt{tag}(\square,a_2) = \mathtt{atx}(a_1) \Rightarrow \mathtt{get}'(a,c,b)$

(3.3) $\mathtt{get}'(a_0,\mathrm{R},b) \wedge \mathtt{isval}(b) \wedge \mathtt{get}'(a_0,\mathrm{L},a_1) \wedge \mathtt{get}'(a,c,a_2)$
$\wedge\ \mathtt{tag}(\square,a_2) = \mathtt{atx}(a_1) \Rightarrow \mathtt{get}'(a,c,b)$

Theorem (getp): $\mathtt{get}(a,c,b) \Rightarrow \mathtt{get}'(a,c,b)$

Proof: We show by induction on p that $\mathtt{anc}(p,l) = a$ and $\mathtt{anc}(p,l.c) = b$ implies $\mathtt{get}'(a,c,b)$. If p is empty the result follows from clause (mt) of the definition of $\mathtt{get}'$. Assume $p = p_0,(r,l_0)$ and $e_{\mathtt{init}} \xrightarrow{p_0} e \xrightarrow{(r,l_0)} e'$. If $r = 1$ there are three cases of interest: (1.1) $l.c = l_0$; (1.2) $l = l_0.\mathrm{L} \wedge c = \mathrm{B}$; and (1.3) $l = l_0.\mathrm{L.B} \wedge c = \mathrm{L}$. If $r = 2$ there are three cases of interest: (2.1) $l.c = l_0$; (2.2) $l = l_0.\mathrm{L} \wedge c = \mathrm{B}$; and (2.3) $l = l_0.\mathrm{L.B} \wedge c = \mathrm{R}$. If $r = 3$ there are three cases of interest: (3.1) $l.c = l_0 \wedge \mathtt{tag}(p_0,l_0.\mathrm{L.B}) \neq \mathtt{atx}(l_0.\mathrm{L})$; (3.2) $l.c = l_0 \wedge \mathtt{tag}(p_0,l_0.\mathrm{L.B}) = \mathtt{atx}(l_0.\mathrm{L})$; and (3.3) $l = l_0.l_1 \wedge \mathtt{tag}(p_0,l_0.\mathrm{L.B}.l_1.c) = \mathtt{atx}(l_0.\mathrm{L}) \wedge \mathtt{tag}(p_0,l_0.\mathrm{L.B}) \neq \mathtt{atx}(l_0.\mathrm{L})$. In each of these cases we use the corresponding clause in the definition of $\mathtt{gen}'$. For all of the remaining possible positions of l relative to l_0 we have $\mathtt{anc}(p,l) = \mathtt{anc}(p_0,l') = a$ and $\mathtt{anc}(p,l.c) = \mathtt{anc}(p_0,l'.c) = b$ where $l' = \mathtt{pred}(e,(r,l_0),l)$. Hence by induction we are done. □

5.1.2. Approximating canbring

The definition of canbring is in fact too restrictive to allow us to express the conditions we need in constructing the approximations canbring′ and doubler′. This is because we want to express not only the possibility of one lambda-node appearing below another, but also the possibility of a variable-node appearing below a lambda-node. To solve this problem we define a larger relation canbring*. $\mathtt{canbring}^*(a_1,a_2)$ holds if either $\mathtt{canbring}(a_1,a_2)$ or a_1 is a value node, a_2 is an atx-node, and there is a rewriting p of $e_{\mathtt{init}}$ such that there is a p-descendant of a_2 which has a p-descendant of a_1 between it and its binding location.

Definition (canbring*): For $a_1,a_2 \in \mathbf{A}$ $\mathtt{canbring}(a_1,a_2)^*$ just if $\mathtt{canbring}(a_1,a_2)$ or $\mathtt{isval}(a_1)$ and there are p, l, c, l_0, l_1 such that $c \in \{\mathrm{R},\mathrm{B}\}$ and $\mathtt{anc}(p,l.l_0.c) = a_1$, $\mathtt{anc}(p,l.l_0.c.l_1) = a_2$, and $\mathtt{tag}(p,a_2) = \mathtt{atx}(l)$.

Lemma (canbring*): $\texttt{canbring}(a_1, a_2) \Rightarrow \texttt{canbring}^*(a_1, a_2)$.

Definition (canbringp): $\texttt{canbring}'$ is the least relation on $\mathbf{A} \times \mathbf{A}$ such that

(mt.i) $a_1 = l.c \wedge c \in \{\mathrm{R}, \mathrm{B}\} \wedge a_2 = a_1.l_2 \wedge \texttt{tag}(\square, a_1) = \texttt{tag}(\square, a_2) = \texttt{lam}$
$\Rightarrow \texttt{canbring}'(a_1, a_2)$

(mt.ii) $a_1 = l.l_1.c \wedge \texttt{isval}(a_1) \wedge c \in \{\mathrm{R}, \mathrm{B}\} \wedge a_2 = a_1.l_2 \wedge \texttt{tag}(\square, a_2) = \texttt{atx}(l)$
$\Rightarrow \texttt{canbring}'(a_1, a_2)$

(3) $\texttt{get}'(a_0, \mathrm{R}, a_4) \wedge \texttt{get}'(a_0, \mathrm{L}, a_3) \wedge \texttt{canbring}'(a_4, a_2) \wedge \texttt{canbring}'(a_1, a_5)$
$\wedge \texttt{tag}(\square, a_1) = \texttt{lam} \wedge \texttt{tag}(\square, a_5) = \texttt{atx}(a_3) \Rightarrow \texttt{canbring}'(a_1, a_2)$

Theorem (canbringp): $\texttt{canbring}(a_1, a_2) \Rightarrow \texttt{canbring}'(a_1, a_2)$

Proof: We will show that $\texttt{canbring}^*(a_1, a_2)$ implies $\texttt{canbring}'(a_1, a_2)$. For this, we show by induction on p that

(i) $\texttt{tag}(\square, a_1) = \texttt{tag}(\square, a_2) = \texttt{lam}$, $\texttt{anc}(p, l.c) = a_1$, $\texttt{anc}(p, l.c.l_2) = a_2$, and $c \in \{\mathrm{R}, \mathrm{B}\}$ implies $\texttt{canbring}'(a_1, a_2)$.

(ii) $\texttt{anc}(p, l.l_1.c) = a_1$, $\texttt{isval}(a_1)$, $\texttt{anc}(p, l.l_1.c.l_2) = a_2$, $\texttt{tag}(p, l.l_1.c.l_2) = \texttt{atx}(l)$, and $c \in \{\mathrm{R}, \mathrm{B}\}$ implies $\texttt{canbring}'(a_1, a_2)$.

If p is empty the result follows from clauses (mt.i,ii) of the definition of $\texttt{canbring}'$. Assume $p = p_0, (r, l_0)$ and $e_{\texttt{init}} \xrightarrow{p_0} e \xrightarrow{(r, l_0)} e'$. If $r \in \{1, 2\}$ then for all allowed positions of l relative to l_0 the result follows by induction. If $r = 3$ then the only interesting case is $\texttt{pred}(e, (r, l_0), l.c) = l_0.\mathrm{L}.\mathrm{B}.l_1'$ and $\texttt{pred}(e, (r, l_0), l.c.l_2) = l_0.\mathrm{R}.l_2'$ for some l_1', l_2'. Then (i) and (ii) both follow from clause (3) of the definition of $\texttt{canbring}'$. □

5.1.3. Approximating doubler

Definition (doublerp): $\texttt{doubler}'$ is the least relation on $\mathbf{A}$ such that

(mt) $l_1 \neq l_2 \wedge \texttt{tag}(\square, a.\mathrm{B}.l_1) = \texttt{tag}(\square, a.\mathrm{B}.l_2) = \texttt{atx}(a) \Rightarrow \texttt{doubler}'(a)$

(3) $\texttt{doubler}'(a_1) \wedge \texttt{get}'(a_0, \mathrm{L}, a_1) \wedge \texttt{get}'(a_0, \mathrm{R}, a_2) \wedge \texttt{canbring}'(a_2, a_3)$
$\wedge \texttt{tag}(\square, a_3) = \texttt{atx}(a) \Rightarrow \texttt{doubler}'(a)$

Theorem (doublerp): $\texttt{doubler}(a) \Rightarrow \texttt{doubler}'(a)$

Proof: We show by induction on p that $\texttt{anc}(p, l) = a$, $l_1 \neq l_2$, and $\texttt{tag}(p, l.\mathrm{B}.l_1) = \texttt{tag}(p, l.\mathrm{B}.l_2) = \texttt{atx}(l)$ implies $\texttt{doubler}'(a)$. Assume $l_1 \neq l_2$, $\texttt{anc}(p, l) = a$ and $\texttt{tag}(p, l.\mathrm{B}.l_1) = \texttt{tag}(p, l.\mathrm{B}.l_2) = \texttt{atx}(l)$. If p is empty the result follows from the clause (mt) of the definition of $\texttt{doubler}'$. Assume $p = p_0, (r, l_0)$ and $e_{\texttt{init}} \xrightarrow{p_0} e \xrightarrow{(r, l_0)} e'$. If $\texttt{pred}(e, (r, l_0), l.\mathrm{B}.l_1) \neq \texttt{pred}(e, (r, l_0), l.\mathrm{B}.l_2)$ then $\texttt{anc}(p, l)$, $(p_0, \texttt{pred}(e, (r, l_0), l))$,

and a are equal. Also $\mathtt{tag}(p_0, \mathtt{pred}(e,(r.l_0), l.\mathrm{B}.l_j))$, $\mathtt{atx}(\mathtt{pred}(e,(r,l_0),l))$, and $\mathtt{tag}(p, l.\mathrm{B}.l_j)$ are equal and the result follows by induction. Thus we may assume $r = 3$, l is a proper prefix of l_0, $\mathtt{pred}(e,(r,l_0),l.\mathrm{B}.l_1) = \mathtt{pred}(e,(r,l_0),l.\mathrm{B}.l_2) = l_0.\mathrm{R}.l'$ for some l', and the result follows from clause (3) of the definition of $\mathtt{doubler}'$. □

5.2. Approximating gen

Definition (genp): $\mathrm{gen}'(a,b)$ just if $a, b \in \mathbf{A}_{\mathtt{lam}}$ and for some $a_0, a_1 \in \mathbf{A}$

$$\mathtt{doubler}'(a) \wedge \mathrm{get}'(a_0, \mathrm{L}, a) \wedge \mathrm{get}'(a_0, \mathrm{R}, a_1) \wedge \mathtt{canbring}'(a_2, b).$$

Theorem (genp): $\mathrm{gen}(a,b) \Rightarrow \mathrm{gen}'(a,b)$

Proof : An easy consequence of (**gen.char**). □

5.3. Usefulness of the approximation

We would like to think of the computable definition of gen′ as a program satisfying a two-part specification: (1) gen′ is safe; (2) gen′ is useful. Formalizing notions of safety is well-understood, but formalizing notions of usefulness is an open problem. At the present we have only some miscellaneous criteria, described in this section.

One criterion of usefulness is non-triviality: the requirement that there exists some expression $e_{\mathtt{init}}$ for which gen′ is smaller than the trivial approximation $\mathbf{A}_{\mathtt{lam}} \times \mathbf{A}_{\mathtt{lam}}$. As mentioned earlier, our definition of gen′ satisfies this criterion.

Another criterion is to require that the program for gen′ compute gen exactly on some test suite of interesting expressions. A finite test suite is hardly a specification, since a trivial program modified to handle the test suite examples as special cases would satisfy the specification. However a good test suite can be useful in identifying problems with the approximation.

Another possibility would be to require that gen′ = gen for certain infinite sets of expressions. For example, our definition of gen′ agrees with gen on expressions that contain no doublers initially.

Lemma (no.doubler): If $e_{\mathtt{init}}$ contains no doublers then gen and gen′ are empty.

Proof : Assume $\mathtt{tag}(e_{\mathtt{init}}, l_1) = \mathtt{tag}(e_{\mathtt{init}}, l_1) = \mathtt{atx}(l)$ implies $l_1 = l_2$ for $l_1, l_2, l \in \mathtt{locs}(e_{\mathtt{init}})$. By safeness it suffices to show that gen′ is empty. Show by contradiction that $\neg\mathtt{doubler}'(a)$ for $a \in \mathbf{A}_{\mathtt{lam}}$. Choose $a \in \mathbf{A}_{\mathtt{lam}}$ with miminal derivation of $\mathtt{doubler}'(a)$. The last rule applied cannot be (mt) by hypothesis. The last rule applied cannot be (3) by minimality. □

6. Towards a general purpose simplifier

The ultimate goal of this work is to develop simplifiers which are of practical use as as automatic program manipulation tools. The work presented here provides a foundation for developing general-purpose expression simplifiers. We have extended the beta-v reduction rule by adding rearrangememt rules that substantially increase the simplification power. These rules remain valid for a wide range of extensions of the lambda calculus by primitive operations to permit embedding of traditional programming languages. We have seen that there are trade-offs between maintaining confluent systems and increasing simplification power. What remains to be done is to work out a variety of substantial examples to test the practical applicability of the various strategies and to determine what are the limiting factors in practical situations. In this section we discuss potential deficiencies and possible improvements of our analysis.

6.1. Approximating gen more accurately

Although for some expressions, the computed gen′ estimates gen exactly, there are other some expressions where the approximation is poor. For example if

$$e_{\mathtt{init}} = \lambda^1 a.\lambda^2 times.\mathtt{let}^3\{twice := \lambda^5 f.\lambda^6 x.f\,(f\,x)\}$$
$$\mathtt{let}^4\{sqr := \lambda^7 x.times\,x\,x\}$$
$$twice\,twice\,sqr\,a$$

then $\mathtt{gen} = \{3 \to 5,6;\ 5 \to 5;\ 4 \to 7\}$ but $\mathtt{gen}' = \{3 \to 5,6;\ 4 \to 4,5,6,7;\ 5 \to 4,5,6,7;\ 6 \to 4,5,6,7;\ 7 \to 4,5,6,7\}$. (Here $3 \to 5,6$ abbreviates $(3,5),(3,6)$, and so on.) Thus the set **B** cannot include any of $4,5,6,7$ and statically-limited rewriting is unable to fully simplify the expression.

One way to improve the simplifier is to more accurately approximate gen. This can be done systematically as follows. For a given expression, define relations $\mathtt{xdoubler}(p,l,a)$, $\mathtt{xget}(p,l,a_1,c,a_2)$, and $\mathtt{xcanbring}(p,l,a_1,a_2)$ which, unlike their finite counterparts, completely describe the rewrite history and location where the relationship occurs. A set of rules can be given which define these relations simultaneously by induction on p. The finite (though perhaps uncomputable) relation gen can be defined exactly in terms of these three potentially infinite relations.

To approximate gen, we choose a function f assigning each pair (p,l) in $\mathbf{Rseq} \times \mathbf{Loc}$ a representation s from a finite set **S**. We then define finite but not necessarily computable relations $\mathtt{ydoubler}(s,a)$, $\mathtt{yget}(s,a_1,c,a_2)$, and $\mathtt{ycanbring}(s,a_1,a_2)$ such that the tuple (s,a) is in ydoubler just if (p,l,a_1,a_2) is in xdoubler for some p,l such that $f(p,l) = s$, and likewise for yget and ycanbring. Finally, we apply f to the inductive rules defining xdoubler, xget and xcanbring to obtain rules defining computable relations ydoubler′, yget′ and ycanbring′ which are guaranteed to be supersets of ydoubler, yget and ycanbring.

The value s in a tuple (s, a) satisfying `ydoubler` is a partial history telling how node a becomes a doubler. In this paper we took **S** to be a one-element set, throwing away the history so that `doubler` could play the role of `ydoubler` (and similarly for `get` and `canbring`) By keeping more history information, it should be possible to approximate gen with arbitrary accuracy; the only drawback would be the increased cost of the calculation. This approach is similar the use of procedure strings and their abstractions in the inter-procedural analysis of Scheme programs [Harrison 1989].

6.2. Alternate non-standard interpretations

gen is itself an approximation to the information contained in the xgen relation. Forgetting the path along which one node generates another when computing generates cycles introduces ficticious cycles – it is possible that $\mathtt{xgen}(p, a, b)$ and $\mathtt{xgen}(q, b, a)$ hold, but never along the same path. Note that this sort of loss of information is avoided by the dynamically-limited strategy. Thus, one could look for better approximations to xgen (that would enable statically-limited rewriting to subsume more of the simplifications allowed by dynamically-limited rewriting. In the example of the previous section, xgen has a cycle although all rewritings from the given initial expression terminate. Thus one might also look for an alternative non-standard interpretation corresponding to a different analysis of the cause of non-termination.

6.3. Preserving context information

We separated simplification from the continuation-passing transformation in order to simplify the basic transformation and to develop a generic simplifier that could be shared among a variety of program manipulation tools. Of course this means loss of information. For example a continuation-passing transformer can carry out beta reductions based on knowledge about whether the application came from the original program or was introduced by the transformation. This approach has been successfully used in developing a continuation-passing transformation program [Danvy, private communication].

We gained simplicity by considering only the language of the pure lambda calculus. Following [Landin 1966] we can represent (by adding primitive constants and syntactic sugar) a wide range of language features (block structure, loops, recursive definition, branching, assignment, goto, escape, labels, ...) without invalidating our reduction rules. In fact any set of rules that are valid in the lambda-c calculus will have this property. Again we lose information in translating from a richer language to the lambda calculus and we may want to consider more refined simplification mechanisms based on richer languages. For example [Moggi 1989] treats `let` as a construct distinct from lambda-application and gives a normalizing system of let-reductions. The system includes the analog of beta-value reduction and many instances (but not all) of our rearrangement rules. It also includes rules such as

$\texttt{let}\{x := e\}x \mapsto e$ which are not derivable in our system. It will require further investigation to determine the relative merits of the two sets of rules (and other alternatives) as the basis of simplification systems.

To improve the usefulness of a generic simplifier a language is needed for expressing information such as that discussed above. One such language is the two-level lambda calculus [Nielson 1988]. Here there are two copies of each syntactic construct. The distinction can be interpreted as compile-time vs run-time or as expressing binding time information [Jones *et al.* 1989]. To account for the wide range of information we need to express will require a more general annotation language.

6.4. Adding new rules

In addition to extending the capabilities of a simplifier by increasing the information and lambda rules available one may also wish to add constants to the language and add corresponding delta-rules. These might include rewriting rules for an abstract data type, rules for conditional expressions, rules for updating operations [Mason and Talcott 1989a], or rules for control operations [Talcott 1989, 1990]. In general the combination of two or more terminating rewriting systems does not produce a terminating system. However, [Breazu-Tannen and Gallier 1989] studies combinations of algebraic term rewriting systems and polymorphic lambda term rewriting and shows that properties such as strong normalization and confluence are preserved for a number of combinations.

Acknowledgements

The authors would like to thank Ian Mason for carefully reading earlier versions of this paper, and pointing out numerous obscurities and mistakes. Conversations with Neil Jones and Olivier Danvy have substantially improved our understanding of issues relating to partial evaluation and abstract interpretation.

This research was partially supported by DARPA contract N00039-84-C-0211.

7. References

Abramsky, S. and Hankin, C. (eds.) [1987] *Abstract interpretation of applicative languages* (Michael Horwood, London).

Barendregt, H. [1981] *The lambda calculus: its syntax and semantics* (North-Holland, Amsterdam).

Bondorf, A. [1990] *Automatic Autoprojection of Higher Order Recursive Equations*, ESOP'90.

Breazu-Tannen, V. and Gallier, J. [1989] Polymorphic rewriting conserves algebraic strong normalization and confluence, in: *16th International colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science **372**, pp. 137–150.

Galbiati, L. and Talcott, C. [1990] A Simplifier for Untyped Lambda Expressions, Computer Science Department, Stanford University Technical Report.

Huet, G. [1977] *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems*, IRIA Laboratory Research Report No. 250.

Jones, N. D., Gomard, C., Bondorf, A., Danvy, O., and Mogensen, T. [1989] A Self-Applicable Partial Evaluator for the Lambda-Calculus, *IEEE Computer Society 1990 International Conference on Computer Languages.*

Jones, N. D., Sestoft, P., and Søndergaard, H. [1989] Mix: A self-applicable partial evaluator for experiments in compiler generation, *Lisp and Symbolic Computation*, **2**, pp. 9–50.

Landin, P. J. [1966] The next 700 programming languages, *Comm. ACM*, **9**, pp. 157–166.

Mason, I. A. and Talcott, C. L. [1989a] A sound and complete axiomatization of operational equivalence between programs with memory, *Fourth annual symposium on logic in computer science*, (IEEE).

Mason, I. A. and Talcott, C. L. [1989b] Programming, transforming, and proving with function abstractions and memories, in: *16th International colloquium on automata, languages, and programming*, Lecture Notes in Computer Science **372**, pp. 574–588.

Moggi, E. [1989] Computational lambda-calculus and monads, *Fourth annual symposium on logic in computer science*, (IEEE).

Nielson, F. [1988] A formal type system for comparing partial evaluators, in Bjørner, D., Ershov, A.P., and Jones, N. D. (editors) *Partial Evaluation and Mixed Computation* (North-Holland) pp. 349–384.

Plotkin, G. [1975] Call-by-name, call-by-value and the lambda calculus, *Theoretical Computer Science*, **1**, pp. 125–159.

Shivers O. [1988] Control Flow Analysis in Scheme, *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation.*

Steele, G. L. [1976] Lambda: the ultimate declarative, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Technical Report 379.

Talcott, C. [1989] *Programming and proving with function and control abstractions*, Stanford University Computer Science Department Report No. STAN-CS-89-1288.

Talcott, C. [1990] A theory for program and data type specification, to appear in: *International Symposium on Design and Implementation of Symbolic Computation Systems (DISCO 90), Capri, Italy* (Springer-Verlag).

Wadsworth, C. [1978] Approximate reduction and lambda calculus models, *Siam J. Comput.* **7** pp. 337–356.

Weise, D. and Ruf, E. [1990] Computing Types During Program Specialization, CSL-TR-90-441, Computer Systems Laboratory, Stanford University.